CORBA IDL/C++ mapping superstructure.

Dyshlevoi K.V.

In this paper some important problems of IDL/C++ mapping from CORBA 2.1 standard are outlined. An approach intended for increasing the convenience and reliability of this mapping usage is suggested.

Introduction

During implementation and debugging of ORB (Object Request Broker) accomplished in ISP RAS (Institute for System Programming of Russian Academy of Sciences) in 1996-1997 considerable experience was obtained in the field of technology of object's interaction in heterogeneous distributed environment. Many shortcomings and advantages of CORBA technology were analyzed from both outside (i. e. from ORB user's viewpoint) and inside (i. e. from ORB developer's viewpoint) of ORB. Numerous problems associated with the standard mapping from IDL (Interface Definition Language) to C++ became obvious. Most of these problems can be formulated as follows: user have to take into account many mapping's rules and restrictions while checking of correctness of user's actions in the most cases is not supported by ORB.

Certainly the question arisen is to propose some mechanism built over standard mapping (i.e. its superstructure) intended to solve these user's problems. The idea suggested is to use so called technique of cover objects. In this case the program system takes upon itself the most aspects of data managing.

Thus let us begin considering the mapping problems which were the reason for suggestions described in detail in the second chapter of the paper.

1. Problems of CORBA IDL/C++ mapping

1.1. One of the main abstractions of CORBA [1] is a notion of object reference. Objects implementing interfaces specified in IDL (Interface Definition Language) can be referred to by a special kind of data (i. e. object references). A name of some interface (for instance, *A*) can be pointed as a type of method's parameter in description of interface's methods. It means that it is possible to pass through corresponding parameter a pointer to object implementing interface *A*.

Because of the different ways an object reference can be used and the different possible implementations in C++ [2], an object reference maps to two C++ types, which quite differ one from the other in the scheme of memory usage. For interface *A*, these types are named *A_var* and *A_ptr.* The pointer type (*A_ptr*) provides a primitive object reference, which semantics is similar to C++ pointer while the object reference variable type (*A_var*) is a special object having constructors and destructor providing ownership of memory it points to. That is the object will automatically release the memory when it is deallocated. Mixing data of types *A_var* and *A_ptr* is possible without any explicit operations or casts, it is possible to assign value of one of these data types to another. In all such assignments pointed data are never duplicated. But assignment correctness is not checked. And it is a widespread source of mistakes in object reference usage. For example, the following sequence of assignment statements is incorrect:

```
{
    A_var var1 = ..; // some initialization
    A_ptr ptr = var1;
    A_var var2 = ptr;
}
```

In all of the assignments memory is not duplicated and three variables (*var1*, *var2* and *ptr*) point to the same data before end of the block. Leaving the block an error of dynamic memory usage appears: calls of destructors of *var1* and *var2* release twice the same memory pointed to by these variables.

Although CORBA standard warns programmer to avoid such situations, similar mistakes appears very often even in small applications. It is because mixing of *A_var* and *A_ptr* types causes necessity to remember about all these variables dependencies during program writing. Besides programmer must remember the scheme of usage of object references as parameters (look at ***1.3.***).

1.2. According to standard mapping requirements programmer should pay special attention to usage of *duplicate* and *release* operations. The *duplicate* operation explicitly copies the object reference and the *release* operation destroys it. Above example (from ***1.1***) can be made correct with use of *duplicate* operation:

```
{
    A_var var1 = ..; // some initialization
    A_ptr ptr = var1;
    A_var var2 = ptr->duplicate();
}
```

It is clear, however, that usage of these operations also make programmer think of relationships between all object references used. Every unnecessary *duplicate* invocation or absent *release* invocation may cause memory of object reference's data to be never deleted. Such a mistake (for instance, in a permanently executing monitor) may entail exceeding of available dynamic memory size. Reverse situation (unnecessary *release* or absent *duplicate*) will rapidly lead to memory usage error as it was shown in the above example.

1.3. Argument passing rules are the most difficult and hardly realized part of the standard mapping.

C++ types used for parameter passing depend both on concrete IDL data type and parameter passing direction (IN, OUT, INOUT and RESULT). Common scheme of argument passing could be deduced only for basic data types denoted as *simple* in Table 1.

Data type	**In**	**Inout**	**Out**	**Return**
simple	simple	simple &	simple &	simple
objref_ptr	objref_ptr	objref_ptr &	objref_ptr &	objref_ptr
struct, fixed	const struct &	struct &	struct &	struct
struct, variable	const struct &	struct &	struct *&	struct *
union, fixed	const union &	union &	union &	union
union, variable	const union &	union &	union *&	union *
string	const char *	char * &	char * &	char *
sequence	const sequence&	sequence &	sequence *&	sequence *
array, fixed	const array	array	array &	array slice*
array, variable	const array	array	array slice*&	array slice*
any	const any &	any &	any *&	any *

Table 1 Parameters and results mapping in CORBA

Second, there is a collection of 6 argument passing cases that user should take into account. These cases are distinguished both in data type and direction of a parameter. Client and server developers must learn well all of these agreements.

1.4. One of the most arguable points concerned with data type mapping is division of data types to fixed-length and variable-length types. A type is variable-length type if it is one of the following types:

- the type *any*,
- a bounded or unbounded string,
- a bounded or unbounded sequence,
- an object reference or reference to a transmissible pseudo-object,
- a struct or union that contains a member whose type is variable-length,
- an array with a variable-length element type,
- a typedef to a variable-length type.

This division of types as shown in Table 1 considerably affects mapping of structure, union and array types. This scheme makes one take into account this type division during writing of server object method's signatures, organizing result returning and invocations of methods.

Certainly, in some cases a little modification of IDL specification (for example, substitution of variable-length type for fixed-length type of an only field of structure) entails change of property "fixed or variable" of whole compound type. Such a change involves re-writing both client and server code. It is clear, that definition of variable-length type is recursive and the single change of the property may involve recursive changes of "fixed-variable" property for all compound types containing the type with the modified property. Of course, it entails re-writing client and server code if OUT and/or RESULT parameters of any of the types with the modified (directly or indirectly) property are used even if originally modified field is not directly used there.

1.5. For every compound data type, for example *T*, mapping provides a class named *T_var* which automatically deletes the pointer when an instance (object) of the class is destroyed. Scheme of usage of such a mediator (*T_var*) is common for all mediator's types and considerably easy for client but mapping doesn't force client to use *T_var* for argument passing. So client can pass and return parameters without the aid of corresponding mediators, therefore client has a considerable chance to make a mistake, forgetting about some of the standard mapping schemes of memory usage (see ***1.3.***).

1.6. For programmer implementing methods of server object all parameters are transmitted without any mediators of *T_var* type. Hence, programmer should precisely know argument passing mapping and meet all parameters' memory management rules. For example, mapping makes programmer explicitly duplicate server object's data to return variable-length result (including OUT parameters) even from *T_var* variable:

```
T  * f_with_res_and_out (A_ptr &obj_ref)  // OUT parameter
{
   A_var out;
   T_var res;
   ...
   // incorrect (but both lines could be successfully compiled!):
   // obj_ref = out;
   // return res;
   //
```

```
    // correct:
    if (CORBA::is_nil (out))
        obj_ref = A::_nil ();
    else
        obj_ref = out->duplicate();
    if ((T*)res == NULL)
        return  new T; // see 1.7 below
    else
        return  new T (*((T*)res));
}
```

1.7. Mapping lists several cases when a *NULL* pointer is not allowed to be passed (IN and INOUT parameters by client) or returned (OUT and RESULT parameters by server). Hence, programmer has to fill parameters even when they are not used by another side. For instance, if some object's method raises an exception then nothing but the information about the exception is returned to client. But even in this case mapping doesn't allow to return a *NULL* reference for not needed OUT and RESULT parameters.

1.8. Filling of objects of *T_var* type is another source of mistakes. Every *T_var* class has *T** constructor and *T** assignment operator (*T_var &operator=(T*)*). In the both operators *T_var* object assumes memory pointed to by *T** parameter. That is, it must be the dynamic-allocated memory that will be deleted either when the *T_var* is destroyed or when a new value is assigned to it. This scheme makes programmer copy data allocated in not dynamic memory into dynamic memory before filling corresponding *T_var.* Using non-dynamic memory for *T_var* filling entails error while deletion of this memory. Striking example of incorrect *T_var* filling is *String_var* filling:

```
CORBA::String_var var = “some string“;
```

Instead of this natural form of assignment the *”some string”* must be explicitly copied into dynamic memory. For dynamic allocation of strings, mapping provides *string_dup()* function defined in the CORBA namespace. Using this function it is possible to make the above example correct:

```
CORBA::String_var var = CORBA::string_dup (“some string“);
```

For the others data types mapping doesn't define functions similar to this one. Hence user have to take care of allocation and filling of the dynamic memory.

1.9. It was remarked in ***1.8*** that *T** constructor creating a *T_var* saves the *T** pointer without deep-copying data pointed to by the pointer. This is the only way to fill

T_var object not from another *T_var* object but directly with data of *T* type. Therefore *T_var* object setting from *const T** can be achieved only by explicit copying *const T** data into dynamic memory.

1.10. The default constructor creates a *T_var* containing a *NULL* value of *T** type. Hence compliant application should not attempt to access and/or return the *T_var* (as OUT and INOUT parameters too) before first explicit assignment of some value to it.

1.11. The problem of incorrect default value exists not only for *T_var* types but for corresponding *T* types too. For instance, the default union constructor does not initialize the discriminator or any of union members. Therefore it is an error for an application to access the union before setting it, but ORB implementations are not required to detect this error due to the difficulty of doing so.

1.12. Especially for insertion array of some type, say *A*, into *Any* mapping provides *A_forany* type which hasn't analogs among other data types. As it is declared in CORBA standard *A_forany* type must be distinct from *A_var* type in memory managing scheme: *A_forany*'s destructor doesn't delete array it points to. The reason for defining such a special type is follows: if the type of array's elements is *B* and *B* type is not array then mapping has already defined insertion operation into *Any* from *B**. But it is known that in C++ (and C) a variable of type *B[]* (i.e. *A*) is used in the same way as a variable of type *B**. Hence, in the sense of argument passing, compiler doesn't distinguish an array from a pointer to its first element. Attempt to implement insertion operation (<<=) into *Any* from *A* causes a conflict with the same operation for *B**.

In spite of class *A_forany* existence, user can make use of insertion operation <<= into *Any* using array as a source and only the first element of this array will be inserted into *Any* instead of the whole array. In such a situation programmer has a good chance to spend a lot of time finding error in the program.

In the end of this chapter it is worth to note that the most part of above problems results in numerous errors in dynamic memory usage while programs writing and debugging. And every C or C++ programmer knows how difficult and "time-wasting" such a debugging is.

2. Superstructure for IDL/C++ mapping

2.1. Cover's usage

Anyone comparing CORBA IDL/C and IDL/C++ mappings can conclude that these mappings are similar and they are based on the common approach. For example, argument passing considerations are almost identical in both mappings: the only difference is that * in C is changed to *&* in C++ somewhere, memory management rules are absolutely equal. And so IDL/C++ mapping doesn't use object-oriented features of C++ in full measure. And all problems pointed out in the 1st chapter can be considered as a result of this politic. Obvious question arising is how to add object orientation to the mapping to solve all these problems.

Let us consider the following extension of the standard mapping. For all compound types (structures, unions, sequences, arrays, strings, object references and *Any*) special class-mediators called **covers** are defined. But in contrast to *T_var* of CORBA mapping, cover provides more reliable mechanism of memory management on the one hand, and a set of explicit operations for user's optimization on the other hand. The main idea of cover approach is that user manipulates data only by means of such covers. In particular, it allows to simplify and increase reliability of arguments passing and getting results (this is a solution of the problem ***1.5***).

Using the mapping's superstructure user doesn't have to take into account the set of the standard mapping memory usage requirements (see problems ***1.3*** and ***1.6***). With cover technique being used the aspects of memory management required for parameter's and result's transfer are hidden from user by the internal covers' mechanisms.

Thus, the extensions rules of argument passing can be described considerably easier than in the standard CORBA mapping (compare the tables depicted in Table 1 and Table 2). These rules avoid the CORBA mapping complication and division data types to fixed-length and variable-length one, solving problems ***1.3, 1.4*** and ***1.6***.

Data type	In	Inout	Out	Return
simple	simple	simple &	simple &	simple
compound (S)	const S_cvr &	S_cvr &	S_cvr&	S_cvr

Table 2 Parameters and results mapping in the superstructure

Consider the scheme of the cover technique in detail. For any compound type *T* the type of cover object named *T_cvr* is defined. There are two modes of cover's usage. The first mode is simple and reliable, and another one is effective, but more complicated. User can choose the scheme of work with data according to his or her needs.

2.2 Basic mechanism

With this approach being used, user can imagine cover as a “smart“ container of data of corresponding compound type. The main idea of memory management of this approach is that the memory contained in the cover is not available for explicit user’s manipulations. That is, the cover creates itself the memory for its data and deletes it. The only actions with cover allowed for user are follows: to copy data to cover’s memory setting entire compound value, to set separate elements (fields) of compound data of the cover, and to examine cover’s contents.

The important feature of covers is that they need not to be explicitly initialized since they contain the following default data. All their elements (fields) having constructor are filled by the default constructors and others are assigned by zero value. This scheme of initialization allows server implementation to return results of method’s invocation (including OUT parameters) without their explicit filling. It solves the problem ***1.7***. The problem ***1.10*** is also solved by the scheme of default initialization. Covers can be used in the same way both after explicit filling and without it.

This scheme also solves the problem ***1.11*** concerned with incorrect union contents before its explicit initialization. Default initialization mechanism for union covers consists in setting of discriminant and union value according to the first field defined in the union’s IDL specification (this scheme is similar to C++ union initialization scheme). If the type of the first union field is a compound type (say *S*) mechanism of default initialization of appropriate cover type (*S_cvr*) will be used.

There are two ways to assign a new value to the data stored in the cover. The source value can be of both *T_cvr* type (another cover of the same type) and *T* type (the only exception here is a cover for object references, work with which is even simpler and it is described below in the very end of subsection ***2.3***). In both cases the cover deep-copies the data into the memory it manages. Deep-copying means creation of a new copy of the source data completely independent from the source. That is, upon the completion of assignment operators both operands exist independently one from another and they doesn’t point to the common data. With this technique being used, ability to assign data of any C++ memory kind *(static, automatic, dynamic)* into a cover is achieved.

Besides, value of a cover can be assigned into a variable of *T* type. Such assignment results in deep-copying too. So in this case cover also behaves like data of type *T.*

To illustrate the suggested assignment scheme consider the simple struct type *S* with the single field of type *long*. Let the four variables are defined and their initial values are specified after double slash:

```
S_cvr       Cvr;          // 1
S           VarData;      // 3
const S     CnstData;     // 3
S_cvr       AnotherCvr;   // 5
```

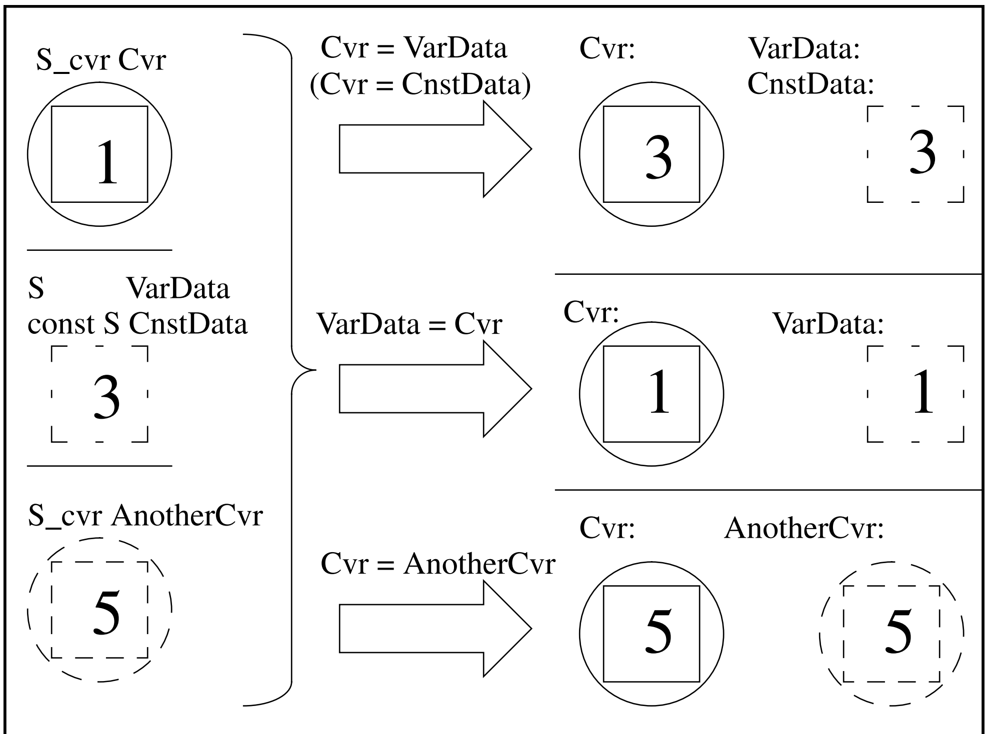
The initial state of the variables is depicted on the left side of Picture 1. Any cover of *S_cvr* type is represented as ellipse, and data of *S* type are represented as square. Shading of figures bounds is used to distinguish memory areas for corresponding data. Picture 1 illustrates data changes which are results of various covers' assignment operations:

1. Cvr = VarData;
Cvr = CnstData;

in both cases data (structured value {3}) from *VarData* or *CnstData* are copied into the *Cvr*'s memory.

2. VarData = Cvr;

data ({1}) from structure contained by *Cvr* are copied into *VarData* variable.



Picture 1 Assignment operations scheme in the base approach

3. Cvr = AnotherCvr;

data ({5}) from structure contained by *AnotherCvr* are copied into the structure contained by *Cvr*.

Besides the assignment operators described above a cover provides operations for access to elements of compound data it contains. Depending on the data type they are:

-> for structures, unions, sequences, object references and *Any*,

[] for strings, sequences and arrays,

<<= and *>>=* for *Any* (i.e. setting/getting operations for basic types to/from *Any_cvr*).

The getting and setting operations *(<<=, >>=)* from *Any* and to *Any* are provided for all types of covers too. Upon completion of these operators memory areas of their operands aren't intersected. Hence, cover-mediator provides required mechanism for conversion between array and Any types in the same way as for all other compound data types, so there is no need for special *T_forany* type and the problem ***1.12*** is completely solved.

Therefore, one can make use of covers not only for argument passing but also for managing contents of compound data types. It is clear that performing all these actions user need not take care about memory contained by the cover.

It should be remarked that the second operand of assignment operator never can be a pointer to data of *T* type (i. e. *T**). Remind that such assignments between variables of *T_var* and *T** types are allowed in CORBA mapping. Consider the example:

```
T  *ptr = ...; // some initialization
T_var  var1, var2;
var1 = ptr;
var2 = ptr;
```

The assignment statements in the example above will result in the both variables of *T_var* type point to the same data, entailing future (not in place of these incorrect action!) dynamic memory usage error. Thus, absence of ability to assign *T** into *T_cvr* solves the problem ***1.8***.

If it is necessary to assign to variable of cover type *T_cvr* (for instance, *Cvr*) a value pointed to by variable of *T** type (for instance, *Ptr*) it is enough to use assignment: "*Cvr = *Ptr*". It is obvious that in the result of the statement both variables are independent (contents of **Ptr* are copied to memory of *Cvr*) and it doesn't matter what classes of memory variable Ptr and the data pointed by it are placed in (solution of the problem ***1.9***).

However, the reverse assignment (i. e. from cover into pointer) is considerably difficult. Let a cover be used as an actual parameter of some C++ function invoked not through an object reference. If the type of corresponding formal parameter is *T** then a cover providing conversion operation to *T** type can be passed as an actual parameter. But there are different ways of parameters' usage inside the function. Consider the following example:

```
void f (T* dust) {
   delete dust; // data pointed by parameter are deleted
}
void g (T* info) {
   cout << info; // only looking at pointed value
}
```

There are two ways to provide implicit conversion mechanism between *T_cvr* and *T** for parameters passing into the both functions (*f()* and *g()*):

1. *T_cvr* to *T** conversion means new dynamic memory allocation and deep-copying of cover's data into it. But after *g(Cvr)* function call the memory was pointed by *info* will be never deleted.

2. This conversion means simply returning of the pointer to the data contained by the cover. Function *g(Cvr)* is completed correctly, while function *f(Cvr)* destroys the memory controlled by the cover. As a result, this memory will be deleted twice. Besides, such a way of conversion contradicts to the agreement that only cover can manage memory allocated for its data.

It is obvious that these both ways don't provide conversion mechanism reliable in any case, therefore conversion operator from *T_cvr* type into *T** type cannot be available.

Certainly, it is possible to implement both functions calls in correct way using only cover's assignment operations:

```
// f() function is called with new dynamically allocated data
T *tmp_ptr = new T (Cvr);
f (tmp_ptr);
//
// g() function is called with data which will be correctly removed
// when block execution is finished
```

```
{
   T tmp_dt = Cvr;
   g (&tmp_dt);
}
```

That is user knowing the semantics of the function's (herein, f() or g()) formal parameter have to explicitly provide necessary actions for parameter passing from the cover into the function. To simplify parameter's passing into user's functions cover's method *T** ***copy()*** is provided. This method allocates dynamic memory, deep-copies data pointed to by the cover to this new memory and returns a pointer to it. The data in the cover are not changed. With this method being used, the *f()* invocation can be performed without temporary variables:

```
f (Cvr.copy());
```

It should be remarked that similar mechanism for correct *g()* invocation requires cover's method returning pointer to the data contained in the cover. However, such a method can't be provided as a part of basic approach since it doesn't meet the requirement of cover's memory independence. The second (advanced) approach defines cover's method *T * look()* for this purpose.

In the end of the first approach description it should be clear that this approach provides for user a simple mechanism represented as covers with transparent memory control. This mechanism can be used for simple and reliable parameters and result passing in CORBA object methods' invocations.

2.3 Advanced mechanism

The main disadvantage of the basic approach is its inefficiency in some cases. It includes a lot of data deep-copying operations and dynamic memory allocations. Of course, for many programs reliability is much more important than execution time. For such applications it is reasonable to use only this first approach. But in some cases program's efficiency is very important too. And it is a reason for additions made in advanced approach described below.

This second approach is intended for C++ programmers who knows well C++ features concerning work with data of different memory classes (static, automatic, dynamic).

Advanced approach simply extends basic approach by adding some new cover's methods for explicit work with memory. In this approach cover is like a "smart pointer" to data which sometimes is manages by user. Covers have more flexible semantics here than cover's constructors and assignment operators of previous approach.

In the context of advanced approach user knows that immediately after default construction cover contains pointer to data with *NULL* value. Default data allocation (as it was described in **2.2**) is implicitly performed only if user attempts to read data from cover with *NULL* pointer. Implicit releasing of data is performed by cover in its destructor: if pointer is not *NULL* and cover owns memory (the only exception are data given by user in *point()* method described below) then pointed data will be deleted.

To check current state of cover's data pointer user can call the following cover's method:

bool ***is_nil*** *();*

This method returns *TRUE* only if the cover contains *NULL* pointer.

For compound type *T* corresponding cover class *T_cvr* has also five following methods for effective explicit covers' memory management (for object reference covers only *release()* method is needed as described below in the end of this subsection):

1. *void* ***assume*** *(T*);*

The cover assumes memory given as an argument. This memory must be dynamic and user should not further delete this memory, since the cover will automatically release it in destructor or in subsequent explicit operation with memory (*assume(), point(), release()*).

2. *void* ***point*** *(const T*);*

This method of cover just saves pointer to data (of any memory type). The pointer assigned in this way will be never released by cover and user should take responsibility of it. It is important to note, the data given to the operation can be changed further independently of user. So, one should be careful when passes data allocated in const memory.

Besides, it is possible to make cover point to automatic memory. In this case user should use the cover only inside of the block where pointed automatic variable is defined. To put data allocated in automatic memory into the cover the following extended form of *point()* operation should be used:

void point (const T, AUTOMATIC);*

For example:

```
static T stat;
T_cvr foo (T &param)
{
   T local;
   T *dyn = new T;
   T_cvr param_cvr, local_cvr, stat_cvr, dyn_cvr, default_cvr;
   dyn_cvr . point (dyn);  // no copy
   stat_cvr . point (&stat);  // no copy
   // ‘param’ and ‘param_cvr’, ’local’ and ‘local_cvr’
   // are defined in the same scope, therefore it is possible:
   param_cvr . point (&param, AUTOMATIC);  // no copy
   local_cvr . point (&local, AUTOMATIC);  // no copy
   ...
   if (...) return param_cvr;  // new data allocation and copy
   if (...) return local_cvr;    // new data allocation and copy
   if (...) return stat_cvr;
   if (...) return dyn_cvr;
   return default_cvr; // ‘default_cvr’ was not explicitly set
}
```

Of course, *point()* operation is dangerous and its usage should be restricted.

3. *T** ***look****();*

By means of this operation user have opportunity to “look” at the cover’s data via own pointer. The cover continues to manage this memory and user should not release pointer taken such a way. In the most general case, this operation returns not *NULL* pointer. If cover is empty it will be automatically initialized and pointer to the created data will be returned. Exception consists in the following case. Method *look()* of *String_cvr* and covers of object reference returns *NULL* if cover is empty, since *String* and object reference are the pointers themselves.

It should be noted that this method can be used for simple passing data from cover to user’s function *g()* from example in **2.2**:

```
g (Cvr.look());
```

4. *T** ***give_data****();*

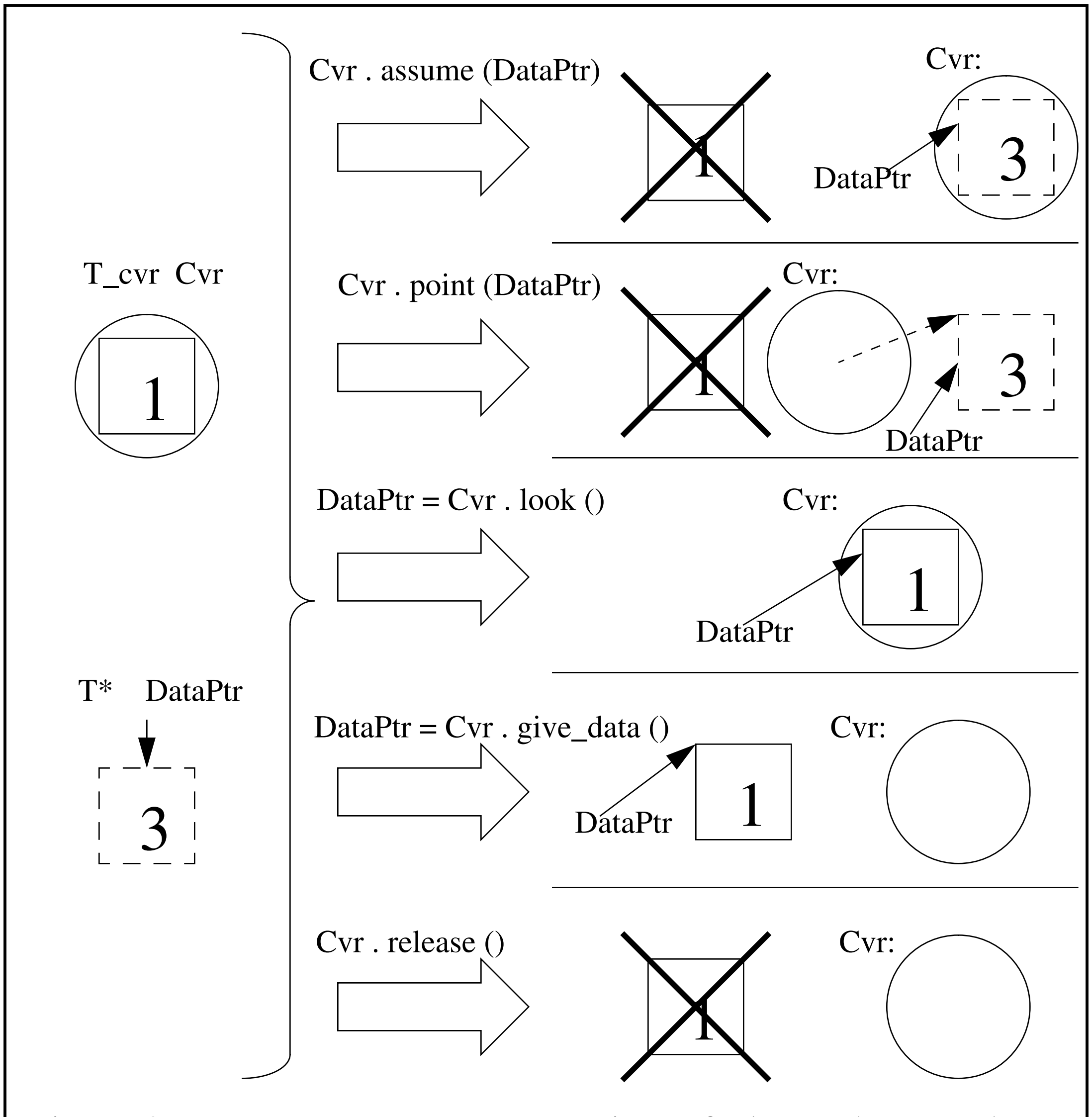
Unlike look() operation, this method “gives up” the memory contained in the cover. In the result of this operation the cover becomes empty and not releases given up memory. So, user should further manage the memory taken this way. If the cover is already empty, new memory will be allocated and pointer to it will be returned.

NULL pointer will be returned only for strings and object references.

5. *void* ***release();***

In the result of release operation the cover will be in the same state as after creation by means of default constructor (cover's pointer will be equal to *NULL*). It does not mean removing of the cover itself.

Picture 2 schematically illustrates all five methods described above. In this picture two intersecting lines mean memory deleting and dashed arrow means that cover points to memory which it can't delete.



Picture 2 Memory management operations of advanced approach

In general outline, each cover contains its own memory and manages it itself. But sometimes it can be useful to have several covers that point to one and the same data. Such situation can be simulated by means of *look()* and *point()* cover's methods:

T_cvr ac, bc;
bc . point (ac . look ());

In this example covers *ac* and *bc* will point to the same data, which was created by means of *ac* cover's initializing mechanism. Ambiguities in memory holding do not appear since *point()* operation do not provide rights of memory management. Common memory will be released only once by cover *ac*.

It is important to remark that the operations described above not only extends covers' functionality but also brings ability to make use data of different C++ memory kinds for argument passing which such a way can be provided in more efficient way than it can be done in accordance to the CORBA IDL/C++ mapping. For instance, CORBA mapping doesn't allow client to allocate memory for OUT parameter of variable-length struct and union, sequence, array and *Any* type. Upon the completion of corresponding method's invocation such a parameter will contain dynamic memory allocated either by server object in the case of local invocation or by ORB in the case of remote one. User should delete this memory later.

Cover's mechanism allows user to allocate its own memory for all OUT parameter's of types listed above. It helps to avoid expenses of allocation and subsequent deallocation of dynamic memory. Consider the example:

MyStruct str;
MyStruct_cvr cvr;
cvr . point (str);
some_object->some_method (cvr); // OUT parameter
cout << cvr->some_field; // result printing

Certainly, server program can make use of the described operations for optimization too. For instance, if server object has to return a variable-length value as OUT parameter, CORBA mapping requires to allocate dynamic memory and to perform deep-copying of the data into this memory. Deep-copying must be done even if returned data are allocated in static memory and accessible without copying after method invocation is finished. Mechanism suggested allows server to return such data without performing all these actions.

For example:

```
void
some_object_implementation::some_method (MyStruct_cvr out_cvr)
{
   static MyStruct static_val;
   MyStruct auto_val;
   ...
   out_cvr . point (&static_val); // saving pointer to static memory
   // or:
   out_cvr = auto_val; // copying from automatic memory
   ...
}
```

Using methods of cover approach both client and server can work with parameters without special dynamic memory allocations. Client can provide own memory for OUT parameter. This memory is used by ORB in the case of remote invocation (i. e. ORB fills this memory with the result obtained from a server process through a net) or by server in the case of local one (client directly sees the result of server's method). Server can fill resulting parameter by means of assignment operation (like assignment from *auto_val* variable in the example). Surely, the server object can make use of static (the *static_val* variable in the example) or dynamic memory. In the case of local invocation and explicit memory management in server object, client can receive the cover containing memory different from previous cover's contents. Therefore, the result obtained should be examined only through the cover rather than through direct usage of the memory previously provided by user.

The questions discussed above was only about increasing performance of parameters passing. Mechanism of result passing was not examined yet. Now consider the following simple example:

```
MyStruct_cvr
some_object_implementation::some_method_with_result ()
{
   MyStruct_cvr  serv_cvr;
   ...
   return serv_cvr;
}
...
// implementation object
some_object_implementation impl;
...
```

// local client in the same process directly calls server implementation:
MyStruct_cvr cli_cvr;
cli_cvr = impl -> some_method_with_result ();

In the example data of server's local automatic variable *serv_cvr* should be transferred into client's variable named *cli_cvr*. The following fact was established during debugging: even in case of local call without ORB usage, C++ compiler doesn't provide direct assignment from *serv_cvr* into *cli_cvr*. After finishing the method a temporary object of *MyStruct_cvr* type is created by the copy-constructor (the source of copying is the *serv_cvr* variable). This constructor allocates memory for storing data in the object created. Then assignment from the temporary object into *cli_cvr* occurs, it is a source of second data deep-copying. Upon the assignment the temporary object is destroyed, with dynamic memory allocated for it being deleted.

In real local or remote object invocation (with ORB usage) such result returning is performed several times in stub (client mediator object) and skeleton (server mediator object). Often necessary "transit" data transfer (i. e. "*return method();"*) is implemented in the same way. In this case C++ compiler also uses scheme based on creating temporary objects.

Of course for really complicated types such a data returning may take much time.

Proposed solution of the problem is to define special *T_res* type. This type serves for data passing between covers with minimal expenses. It helps to optimize returning of result from the method and is used only in operation signatures. As well as *T_cvr*, object of *T_res* type contains a pointer to appropriate type. Unlike *T_cvr* *T_res* type does not overload operators for data members manipulations (*->*, *[],* and so on). The *T_res* type is intended to be used only in operation signatures and nowhere else (for more details see [3]).

Now, if there is a need to optimize returning of result of *MyStruct* IDL type it is possible to use *MyStruct_res* as result's C++ type instead of *T_cvr*. For example:

```
MyStruct_res
some_object_implementation::some_method_with_result ()
{
   MyStruct_cvr serv_cvr;
   ...
   return serv_cvr;
}
```

Note that only the method's signature is changed while method's body remains unchanged. But now for returning result C++ compiler uses a temporary object of

MyStruct_res type pointing to the same data as the *serv_cvr* variable (the only exception is data in automatic memory, given by user in *point (..., AUTOMATIC)* operation, in such a case temporary *T_res* object will contain a replica of these data). With reference counter being used, such assignment and destruction of automatic variable *serv_cvr* result in passing memory ownership from *serv_cvr* to a temporary object. In the example with local call ownership is passed then from temporary object to *cli_cvr*. So value is returned without memory allocations and removing and without data deep-copying (in case when *serv_cvr* contains data in automatic memory, needed allocation and copying are performed only once).

It is important to remark that covers approach allows not to save method invocation's result as it is done in C++ (regardless of *T_cvr* or *T_res* was used). The memory returned will be deleted under temporary cover's destruction if *serv_cvr* owned this memory. In CORBA mapping if variable-length result is not saved then its memory will be never deallocated.

It is need to be remarked that everything said above is based on the assumption that creation and destruction of both covers and data they point to are available to user. However, object reference is the only exception from this statement. The reason is that object reference notion is directly concerned with the ORB and meaningless out of ORB's context. Object reference mechanisms and its contents are transparent for user. Moreover, user can't create and delete data of object reference type. Therefore, user works with object references only via covers and even pointers to object references are not available. Using such covers user needs only assignment operations between covers from basic approach, *release()* and *is_nil()* methods from advanced one (method *copy()* from the first approach is not needed for such covers). For object references *T_res* type coincides with *T_cvr* type.

It is very important to note that *narrow()* method of object references from standard mapping is not used by user of covers technology because of all needed reference transformations are performed by covers:

```
A_cvr a;
B_cvr b;
a = b;
if ( (b . is_nil () == FALSE ) // 'b' contains object reference
   && (a . is_nil () == TRUE) ) // but 'a' doesn't contain anything
  {...} // 'b' value can not be assigned to 'a'
```

Assignment between object reference covers of different types is correct. At runtime after such assignment performed cover *a* will contain not *NULL* reference only if object pointed by reference contained in *b* implements interface *A* (real interface of object pointed to by this object reference either coincides with *A* or is inherited from *A*).

Thus, problems of object reference usage recognized in the beginning of the paper (the problems ***1.1*** and ***1.2***) are solved by covers. Covers perform all needed actions for managed object references themselves (the only memory management method *release()* in such covers can be used to reject object reference before cover deleting or reassignment).

Conclusion

The first approach to cover's usage completely solves problems listed in the beginning of the paper and the second approach provides more efficient and flexible data management mechanism but requires better knowledge of C++ language. User can choose between these two approaches according to his or her needs.

It is very important to note that the proposed technique doesn't pretend to change the standard. This technique is implemented as a superstructure of CORBA IDL/C++ mapping. This superstructure is intended to be applicable to any C++ ORB fully compliant with CORBA 2.0. In general this technique gives user simplicity and reliability in ORB using. Note that efficiency improvement in comparison with standard mapping can be achieved only by using "native" ISP ORB since ORB should work with memory contained by covers directly.

What is about possible place and role of this technique?

Of course, this technique can be added even to already existing CORBA-applications to improve their efficiency (using ISP ORB and advanced approach) or to provide extensibility if complexity of application is planned to be increased (it is well known that it is very difficult to extend applications based on not reliable technique).

The technique can become more convenient both for writing programs and for their debugging while developing applications from the beginning.

Also, the advanced approach included into this technique with its flexible mechanism for work with memory is more convenient and reliable for addition of CORBA mechanisms to already existing C++ applications not oriented to CORBA (some variant of legacy problem).

References

1. The Common Object Request Broker: Architecture and Specification. Revision 2.1, August 1997.
2. Bjarne Stroustrup. The C++ Programming Language, Second Edition. Murray Hill, NJ: AT&T Bell Laboratories, reprinted 1992
3. Dyshlevoi K.V, Solovskaya L.B. Specification of Addition to CORBA IDL C++ Mapping, ISP RAS document. Moscow, 1998.